COMSC-210 Lecture Topic 2
Overloaded Operators, Templates, and Abstraction

Reference
Childs Ch. 2

Overloaded Operators
defining what less-than and equals mean for objects
  e.g., `if (myCar < yourCar)`
why not just use functions???
  e.g., `if (myCar.isCheaperThan(yourCar))`
  because: the *STL* requires it

```
#include <queue>
#include <string>
using namespace std;

struct Semester
{
  string season;
  int year;
};

bool operator<(const Semester& a, const Semester& b)
{
  if (a.year < b.year) return true;
  if (a.year > b.year) return false;
  if (a.season == b.season) return false;
  if (a.season == "SP") return true;
  if (a.season == "SU") return b.season == "FA";
  return false;
}

int main()
{
  Semester a = {"Spring", 2014};
  ...
  priority_queue<Semester> semesters;
  semesters.push(a); // expects < defined
  ...
}
```

Common Overloaded Operators
less-than and equal-to (comparison operators)
stream-insertion (for use with cout)

```
bool operator<(const Car&, const Car&);
bool operator==(const Car&, const Car&);
ostream& operator<<(ostream& out, const Car&);
```

assignment operator (later...)
`const` keyword, later

Stand-alone Overloaded Operators In H/CPP Files
write prototypes in H, AFTER `};` and ABOVE `#endif`
write definitions in CPP, WITHOUT `::` scope resolution

Overloaded Operators As Member Functions
may be programmed as members *or* stand-alones
  per the programmer's preference
e.g., as a member function:

```
bool Semester::operator<(const Semester& other) const
{
  if (year < other.year) return true;
  if (year > other.year) return false;
  if (season == other.season) return false;
```

The Template Declaration Statement
it's NOT a container -- no end marker; just a header
`template <class T>`
(or `template <typename T>`)
...goes *above* each class and function definition
Translation: Childs: `DataType`, Comsc-210: `T`

Using The Template Variable
we use `T` as the variable name
e.g., in a class definition (e.g., an array)
  `T array[10];`
e.g., in a function definition (e.g., a parameter)
  `void putValue(int index, const T& value)`
  `T& getValue(int index)`
treat `T` like it is an *object* as read-only reference

The Object Declaration Statement
need to tell the templated class what is the data type
e.g., `Array<int> a;` replaces `T` with `int`

A Templated Class' H File: Checkbook.h

```
#ifndef Checkbook_h
#define Checkbook_h

template <class T>
class Checkbook
{
  public:
  void setBalance(float amt); // a setter
  bool writeCheck(const T& amt); // a setter
  ...
};

template <class T>
bool Checkbook<T>::setBalance(float amt)
{
  ...
} // yes, even though T is not used!

template <class T>
void Checkbook<T>::writeCheck(const T& amt)
{
  ...
}
...
#endif
```

Variation: write separate H and CPP for templated classes
...but do NOT compile the CPP:
  compiler would not know what to do...
  ...because no data type is specified!

CheckbookDriver.cpp
instead of this: `Checkbook cb; ...`
use this: `Checkbook<float> cb; ...`

testing templated functions
  compiler may not even look at a template that's not used
  so it's possible compile "successfully" with syntax errors
test *ALL* templated functions in the test driver,'
  whether you use them in an app or not!

```
  if (season == "Spring") return true;
  if (season == "Summer") return other.season == "Fall";
  return false;
}
```

Note the `const` keywords and the pass-by-reference
req'd for use with the STL

Member Overloaded Operators In H/CPP Files

write prototypes in H, INSIDE the `struct` or `class` definition
write definitions in CPP, WITH `::` scope resolution

Objects & Arrays As Data Members

```
struct CarDriver
{
  string name;
  License license;
  CarType car[2];
};
```

note the fixed-size array!

Class Templates

class specification and implementation files
with one of the data type specifications
as a *variable*
what's the application???

consider an "array class" (chapter 5)
an array of what?
either program has to be specific
...or... make the data type variable!

Abstraction And ADTs

purpose: ignore the details -- it just works!
e.g., even without rewriting or deriving a subclass
or recompiling, the statement
`priority_queue<Semester> semesters;`
builds a priority queue of `Semester` objects!
templated classes are *abstract data types*
they provide the detail independent of the
actual data type in the application

STL Container Templates

"sequence" containers: vector, deque, list
"associative" containers: set and map
"adaptor" containers: stack, queue

The STL `stack` Template

an *adaptation* of a sequence container
including the stack library
creating a stack: `stack<int> s;` // an empty stack
`s.push(100);` // put the int 100 on the top of the stack
`int x = s.top();` // return a copy of the int on top of the stack
`s.pop();` // delete the int on top of the stack
check if the stack is empty: `s.empty();`
check stack size: `s.size();` // #of ints in the stack

the **"copy-pop"** method
...for inspecting stack contents:
make a copy of the stack
loop through copy with top and pop...
...until copy is empty

```
for (stack<int> copy = s; !copy.empty(); copy.pop())
  ...copy.top()...
```

application to lab 2c RPN calculator...

How to confirm comparison operators in a test driver:

```
cout "Testing operator=="
Create a copy (y) of an existing object (x)
if x == y
  cout "x==y, as expected"
else
  cout "test failed!"
assert(x==y)

Use a setter to modify y so it does not equal x
if x == y
  cout "test failed!"
else
  cout "x not equal to y, as expected"
assert(!(x==y))

cout "Testing operator<"
Use a setter to modify y so it is greater than x
if x < y
  cout "x<y, as expected"
else
  cout "test failed!"
assert(x<y)

Use a setter to modify y so it is less than x
if x < y
  cout "test failed!"
else
```

```
      cout "y not less than x, as expected"
    assert(!(x<y))
```